レガシーな組込みソフトウェアの改良支援ツール

阪井　誠[†]　久保田益史[††]　沖田　昌也[††]　松本　健一[†††]
鳥居　宏次[†††]

A New Environment for Improving Legacy Software on Embedded Systems

Makoto SAKAI[†], Masunori KUBOTA[††], Masaya OKITA[††],
Ken-ichi MATSUMOTO[†††], and Koji TORII[†††]

あらまし　本論文では，改造の繰返しにより保守性が低下した組込みソフトウェアを対象として，プログラムのモジュール構造を評価する方法を提案するとともに，提案方法に基づき開発したソフトウェア改良支援ツールについて述べる．提案する評価法は，構造化分析法におけるモジュール間結合度とモジュール凝結度の概念を応用し，組込みソフトウェアで多用される外部変数によるデータの受け渡しを含めたモジュール構造の改良を支援するものである．また，支援ツールは，(1) 外部変数によるデータの受け渡しを上位関数とのインタフェースとして表示することで，従来のツールでは表示されなかった「階層化されたモジュール間でのデータの流れ」を表示できる，(2) オープンソースのツールを利用することで比較的少ない工数で実現されている，(3) 利用者の経験度を問わずソフトウェア改良法を短時間で習得できる設計ガイドとチェックリストが整備されている，といった特徴をもつ．本ツールを実プロジェクトに導入した結果，保守性低下の原因となっているプログラム構造を発見，改良するのに有効であることがわかった．

キーワード　組込みソフトウェア，レガシーソフトウェア，メンテナンス，リエンジニアリング

1. まえがき

機能追加の目的で改造が繰り返される間に，ソフトウェアの保守性は徐々に悪くなる．新しい機能を付加したソフトウェアを，より信頼性を高く，より少ない工数で，より短期間に開発するために，既存のソフトウェアが改造されることは多い [11]．このような改造が繰り返されると，保守性は徐々に悪くなり，遺産的 (Legacy) ソフトウェア（以下レガシーソフトウェア）と呼ばれるようになる．

組込みソフトウェアは，高い信頼性が求められるほか，タイムリーに製品を市場に投入することが求められるので開発期間短縮の圧力が強く [14],[15]，マーケティングの観点から予算の制約も厳しいという特徴がある．そこで，再利用を行う際にも，流用を増やし，改造をより少なく局所的なものとするため，保守性が悪くなりやすい．

また，組込みソフトウェアには外部変数によるデータの受け渡しが用いられている場合が多く，更に保守性を悪くしている．ソフトウェアが組み込まれる機器は，多くの動作モードや障害時の復旧対策が必要なので，通常の主記憶領域とはメモリ空間が異なる不揮発メモリが用いられる．これらは，実装の容易さや処理速度の観点から，外部変数として実装し，リンク時に不揮発メモリに割り当てられている場合が多い．また，OS とハードウェアの制約からタスク間の高速なインタフェースとして外部変数が用いられる場合もある．このような外部変数は，機能が追加されると動作モードや障害処理が増えるので，更に増加する．外部変数を用いたデータの受け渡しは，旧来から悪いインタフェースであるといわれている [10]．しかし，やむを得ず外部変数を用いる場合に，プログラム構造のどのような点に注意すべきかが示されていなかったことが，保守性を悪くする一因になっている．

このようなレガシーソフトウェアの保守性を向上す

† （株）SRA 先端技術研究所，東京都
SRA Key Technology Laboratory, Inc., 3–12 Yotsuya, Shinjuku-ku, Tokyo, 160–0004 Japan
†† オムロン株式会社，草津市
OMRON Corporation, 2–1 Nishikusatsu 2–chome, Kusatsu-shi, 525–0035 Japan
††† 奈良先端科学技術大学院大学，生駒市
Nara Institute of Science and Technology, 8916–5 Takayama-cho, Ikoma-shi, 630–0101 Japan

るには，外部変数によるデータの受け渡しを用いたソフトウェアのための新しい評価法が必要である．文献[10] 及び文献 [16] では，プログラムの評価法を示し，その評価法を満たすプログラムを設計することで保守性の高いプログラムを実現しようとしている．しかし，これらの方法論では外部変数によるデータの受け渡しは良くないとされており，外部変数を用いざるを得ないプログラムを改良する方法は示されていなかった．

また，効率良く改良するためには，作業を支援するツールが必要である [5]．しかし，従来のツールは従来の方法論に基づいているので，文献 [6] に見られるように外部変数をどの関数が参照しているかを示す機能のみであった．レガシーな組込みソフトウェアの改良には，新しい評価法に基づいたプログラムの評価が可能となり，問題点の発見や改良方法の検討が容易なツールを開発することが必要である．

本論文では，まず，レガシーな組込みソフトウェアを調査して保守性の悪い部分を分析し，評価法を提案する．次に，提案方法に基いて開発したソフトウェア改良支援ツール [18] に関して述べる．提案する評価法では，外部変数によるデータの受け渡しを引数と同じように扱うことができる．支援ツールを用いると，構造化設計の評価法 [16] を応用した改良が可能となる．また，オープンソースプログラム（以下，オープンソースと呼ぶ）であるツールを組み合わせて開発することで，比較的少ない工数で実現されている．更に，利用者の経験度を問わずソフトウェア改良法を短時間で習得できるように，設計ガイドとチェックリストが整備されている．本ツールを実プロジェクトに導入した結果，コードレビューで指摘できなかった保守性低下の原因となっているプログラム構造を短時間で指摘することができたほか，改良後の品質維持にも有効であった．以下，**2.** で関連研究，**3.** で提案する評価法，**4.** で開発したツール，**5.** で結果と考察，**6.** でまとめを述べる．

2. 関 連 研 究

2.1 プログラムの評価法

レガシーソフトウェアには，柔軟に保守できるプログラムであるかどうかがわかるような評価法が必要である．保守には障害を少なくする修理保守，性能・機能・保守性などを改良する完全化保守，環境の変化に合わせる適応保守がある [13]．修理保守を対象とする場合は，文献 [20] に示されるように，欠陥数と相関の高い評価法が必要である．完全化保守や適応保守には，プログラムの変更に対し，柔軟に対応できるような評価法が必要である．レガシープログラムを生じさせるのは，完全化保守であるので，プログラムの新規作成時に用いられるような評価法が有効であると考えられる．

構造化設計では，理解しやすく，変更が容易なプログラムを作成するために，モジュール間のインタフェースを示す「結合度」，同一モジュール内の機能的連結の強さを示す「凝結度」，保守性を高めるための「追加の設計ガイド」で設計を評価する [16]．構造化分析/構造化設計はオブジェクト指向の基本になっているものであり [1]，これらの考え方は現在でも非常に有効な考え方であると考えられる．

結合度は二つのモジュール間の関係度合を示しており，狭く，直接的で，局所的で，明確で，柔軟な連結にすることを目指している．正常な結合として，単純なデータをパラメータとする「データ結合」，複合データを渡す「同一データ結合」，他のモジュールの内部ロジックを制御する「制御結合」の三つがある．良くない結合として，外部変数によるデータの受け渡しを行う「共通結合」，他のモジュールの内部を参照する「内容結合」などがある．なお,「データ結合」であってもパラメータが 20 を超えるような難解なモジュールは凝結度が疑わしいとされている．また,「共通結合」に似ていても，データベースを用いる「データベース結合」は，不揮発性であり，方法論に基づき，一度の処理では必要なデータのみ操作する点で「共通結合」とは異なるとされている．

「凝結度」はモジュールの強さを表し，ただ一つの問題に関連した仕事をこなす「機能的」，あるモジュールの出力が次のモジュールの入力に次々につながる「逐次的」，同一のデータを扱う「通信的」，順に実行される「手順的」，同一のタイミングで実行される「一時的」，同一のカテゴリー処理の「論理的」，関係のない処理の「偶発的」に分けられ,「機能的」が最も良く,「逐次的」と「通信的」までが保守しやすく，他は保守が困難なモジュールとされている．

「追加の設計ガイド」は,「結合度」と「凝結度」だけでは不十分な点を補足し，保守性を高める指針である．具体的には，トップダウン設計，機能を重複させない，小さく，汎用的に，上位の管理モジュールと下位の作業モジュールを分けるといった，評価の基準としてモジュールの機能分解の基準を示している．

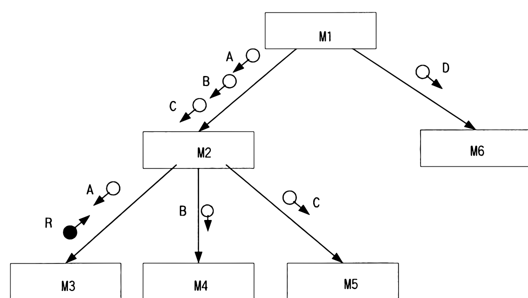

図 1 構造図を用いた評価
Fig. 1 Evaluation using the structured chart.

図 1 は従来の方法論で用いられる構造図である．構造図では上位と下位の呼出関係を矢印で，渡されるデータを丸のついた矢印で表す．図 1 は比較的保守性の良い構造を示しているが，保守性の悪い構造の場合，それはデータの流れに現れる．仮にモジュール M1 が最上位の関数でなく，変数 A,B,C,D がすべて引数で渡されたなら，M1 のインタフェースはやや複雑なものになる．変数 A,B,C を M1 が操作しないなら，M2 以下の処理とそれに関係するモジュールだけで操作するデータであり，それらを構造体へ統合し，インタフェースを単純化すべきかを検討する必要がある．次に，モジュール M3 は M4 と M5 に共通の前処理であるが，独立した共通モジュールにならずに M4 と M5 に M3 の処理が組み込まれていた場合，M4 と M5 のインタフェースに変数 A が共通に現れ，インタフェースはより複雑になり，共通モジュールを抽出するきっかけになる．

オブジェクト指向においても上記の評価法は用いられており，現在でも有効な評価法であると考えられる．「結合度」と「凝結度」によるモジュール化は，オブジェクト指向におけるパッケージングの際に用いられている [4]．また,「追加の設計ガイド」もトップダウン設計を除いて再利用のための作法として用いられている [17]．これらの評価法は一般的なものであり，より良いモジュール化が可能になると考えられる．

2.2 レガシーソフトウェアの改良

過去のソフトウェアをもとにして新しいソフトウェアを作る方法は，プログラム全体を再構築するリエンジニアリングとして研究されてきた．従来の研究には大きく分けると二つの方法があり，トップダウンにプログラム全体を作り直す方法と，レガシーソフトウェアから再利用可能な部分をコンポーネントとして取り出して，それをもとにボトムアップにシステムを再構築する方法である [2]．いずれの場合も専用の分析ツールやナレッジベース等を用いて，ドメインエキスパートと開発者が協力しながらプログラム全体の解析と比較的大規模な開発を行う必要があった．

しかし，限られた予算の中では，プログラムの部分的な改良しか行えない．ソフトウェアのリエンジニアリングを行う場合，改良により得られる効果と開発のリスクを考慮しなければならない [19]．今後の市場性，競合他社の動向，利益等を考慮して予算を決める必要がある．限られた予算の中で改良を行うには，プログラム全体を改良するのではなく，特に問題のあると思われるところに限定して改良すべきである．文献 [9] ではプログラムの規模や分岐の数といった尺度を用いて改良する範囲を限定している．しかし，外部変数が用いられることは考慮されておらず，組込みソフトウェアに適用可能であるかを調査する必要がある．

また，常に高い保守性を維持するには，プログラミングを行った部分を常に見直しておくことが必要である．12 のプラクティスを強調して実施する XP (eXtreme Programming) においては「リファクタリング」というプログラムの改良を常に行うことで，保守性を維持している [3]．このように徐々に改良する方法は，一度に大きな費用が発生しないので，投入可能なコストが少ない．多くの改造が行われているレガシーな組込みソフトウェアでは，改良された部分が徐々に広がるので，有効な方法であると考えられる．

3. 評 価 法

本研究では，レガシーな組込みソフトウェアの保守性を向上させる目的で，外部変数によるデータの受け渡しを行うプログラムの評価法を提案する．本研究では，既存のプログラムの保守性の低下した部分を調査した上で，従来の評価法を拡張した．

3.1 レガシーソフトウェアの調査

3.1.1 レガシーソフトウェアの調査方法

組込みソフトウェアには他のソフトウェアにはない特徴があるため，既存の方法論のみではプログラムを改良することはできない．そこで，レガシーソフトウェアを調査し，保守性の低下したパターンを以下の順に調査した．対象は金銭の入出力を行う端末機器のレガシーソフトウェアである．C 言語で開発されており，6748 ファイルから構成されていた．これらのうち，保守が困難な部分を対象ソフトウェアのエキスパート

にインタビューし，それらを中心に調査した．調査は実行ステップだけでなく，コメントアウトされた過去のソースコードや修正履歴も調査した．

STEP 1　インタビュー

まず，保守性の低下したところはどのような部分であるかを，エキスパートにインタビューした．保守の困難なところは，改造が繰り返された部分であり，入力データや外部変数などに対して，多くの判定が必要な部分であった．条件分岐の多いところの保守性が悪いという評価は文献 [9] と同じであった．

STEP 2　条件分岐を多く含む関数の調査

インタビューの結果をもとに条件分岐を多く含む関数を調査した．条件分岐を多く含む順に並べ換えた関数のリストをエキスパートに示したところ，保守が困難な部分はすべてリストの上位にあったが，条件分岐を多く含む関数すべての保守性が低下しているとはいえなかった．

STEP 3　保守性の悪いプログラムの調査

条件分岐を多く含むプログラムのうち，保守性の低下したプログラムとそうでないプログラムを比較したところ，保守性の低下したプログラムには，以下の特徴があった．

- 多くの外部変数を参照する．
- モジュール構造が複雑である．

3.1.2 外部変数

条件分岐を多く含むプログラムのうち，保守性の低下したプログラムは，多くの外部変数を参照するという特徴のものがあった．機能が追加された際に，追加された処理に必要となる動作モードの設定や障害復旧対策のために，各関数で参照する外部変数が徐々に増えたのである．保守性の低下した関数は単純な機能を実現する関数ではなく，(1) 複雑な判定を行う，(2) 多くの外部変数を判定し順番に処理を実行する「手順的な凝結度」[16] で，複数の機能が一つの関数になっている，という特徴があった．(1) は徐々に増加した外部変数がまとめられずに，多くの外部変数を参照していた．(2) は手順的強度であっても比較的単純であった関数が，長年にわたる機能追加により徐々に複雑になったものである．特に (2) では，機能が追加された際に，既存プログラムのコピーを改造して類似した関数を作るということも行われていた．既存のプログラムの再テストを避けることで，期間短縮を図っていたのである．

3.1.3 モジュール構造

条件分岐を多く含むプログラムのうち，保守性の低下したプログラムには，モジュール構造が複雑であるという特徴のものもあった．単純でない比較的上位の関数を再利用して，共通ルーチンとして使っていた．モジュールを再分割しないで，コードの共有化が行われており，再帰処理になっているところもあった．これは，既に実績のあるソースを用い，既存の処理への影響を最小限にすることで，期間の短縮やコストを削減しようとしていたのである．

このような，保守性の低下した部分は改良される機会はあったが，正しく改良されなかった．システムの大規模な見直しの際には保守性の低下した部分を含めて改良が試みられている．対象としたソフトウェアは文献 [7] と同じようにカスタマイズで追加された新機能を標準化し，複数バージョンの管理を減らす努力が行われている．しかし，機能の統合は行われるものの，外部変数でデータを受け渡すプログラム構造全体を鳥瞰（かん）した上で，ソースコードから保守性の低下した部分をもれなく判定することは難しかった．

すなわち，以下が不足していたので，保守性が改善されず，レガシーソフトウェアになったと考えられる．

- 外部変数によるデータの受け渡しを用いたプログラムを評価する方法論．
- プログラム全体を鳥瞰しながら，方法論の実施に必要とされる情報を表示するツール．

3.2 レガシーな組込みソフトウェアの評価法と改良法

本研究で提案する評価法の基本的なアイデアは，外部変数によるデータの受け渡しを引数と同じように扱うというものである（図 2）．引数が外部変数と違う点は，以下の 3 点である．

- インタフェースが関数の定義として可視化される．
- 引数は上位の関数に与えられるので，同一の引数を使用する関数がグループ化される．
- スタックとして実装されるので，名前渡しの場合は，データが破壊されない．

対象の関数及びその下位の関数で使用している外部変数を関数のインタフェースとして表示することで，3 点目のデータの保護は実現できないものの，引数と同じ方法論を用いることができることになる．**2.1** で示したように「データベース結合」は，不揮発性であり，方法論に基づき，一度の処理では必要なデータのみ操

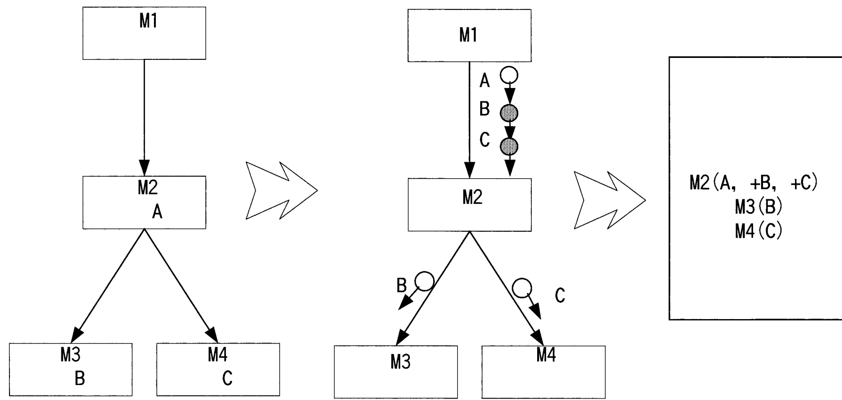

図 2　基本的なアイデア
Fig. 2　The basic idea.

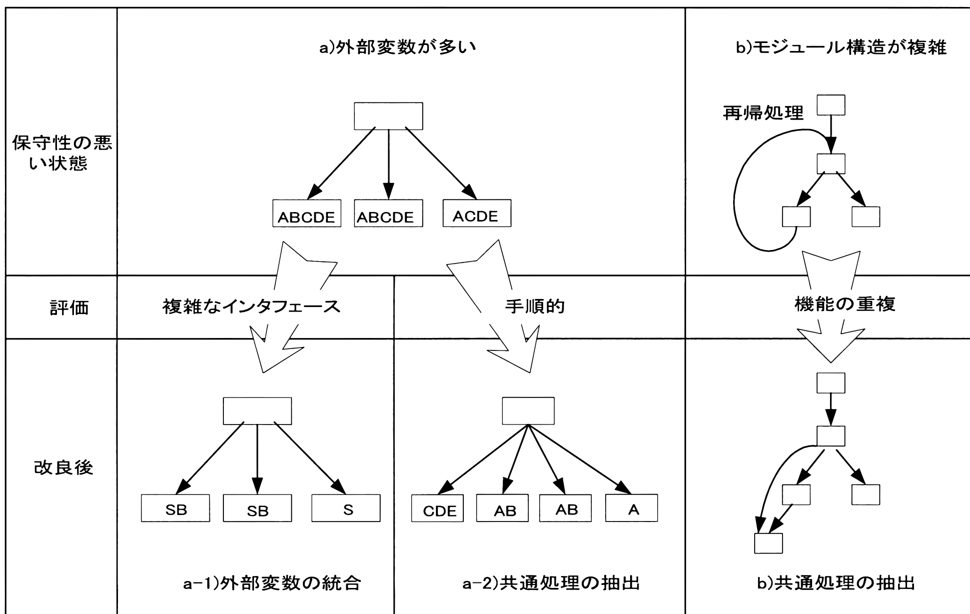

図 3　評価と改良法
Fig. 3　The evaluation and the improving method.

作する共通結合である．組込みソフトウェアで用いられる外部変数は不揮発性であり，引数に関する方法論は文献 [16] を用いることで，参照するデータごとにグループ化（局所化）され，小さく汎用的なモジュール（関数）を実現できるようになる．すなわち，データベースは用いないもののデータベース結合に近い形になり，より保守の容易なプログラム構造になる．

本研究で提案する評価法に基づくと，**3. 1** で挙げた問題は図 3 のように改良できる．A-S は各関数で参照する外部変数，四角は関数，細線は呼出しを示す．多くの外部変数を参照している場合には，外部変数の統合が必要な場合 a-1) と，共通処理を抽出する必要のある場合 a-2) がある．参照されている外部変数がどのような分類に属するかで評価が変わり，改良方法が変わる．同一の分類の外部変数が多い場合は，それらを構造体に統合し，よりわかりやすいインタフェース

に変更できる可能性がある．また，異なる分類の外部変数が多い場合には，外部変数の判定処理を抜き出して，共通処理の処理として抽出できる可能性がある．また，図 3b) のように，再帰処理で実装されたために，モジュール構造が複雑になった場合には，上位関数の共通処理を抽出して，新しい共通関数を作成し，上位関数と下位関数を新しい共通関数を呼び出すように改造することで，モジュール構造を単純にできる可能性がある．これは文献 [3] で “the Once and Once rule” と呼ばれるリファクタリングと同じ方法である．

これらの評価を行って改良するには，外部変数の参照関係を引数と同様に表示するだけでなく，全体を鳥瞰でき，構造体への統合をシミュレートできる必要がある．構造図は大規模になると作成や管理の負担が増えるだけでなく，全体を見渡すことが難しくなる．そこで，図 2 の右に示すような関数の引数形式での表示が有効である．ただし，本来その関数で扱わない外部変数は区別する必要がある（図 2 では＋マークを付けている）．また，構造体への統合は修正範囲が全体に及ぶので，実際の修正が困難な場合がある．そこで，外部変数を実際に構造体に統合せずにどのような結果になるかを表示できれば，全体の構造がわかりやすくなり，共通処理の抽出など他の改良が容易になると考えられる．

4. ツ ー ル

4.1 ツールの設計と実装

ソフトウェア改良支援ツールは，以下の機能を実装している．

- 外部変数によるデータの受け渡しを上位関数とのインタフェースとして可視化できる．
- 外部変数を構造体として統合した場合のシミュレーションができる．
- 全体を鳥瞰しやすいように，プログラム構造をシンプルに表示できる．

具体的には，外部変数を構造体に統合する場合の定義を記述した「外部変数分類定義」と C 言語の「ソースファイル」を入力とし，外部変数をグループ化し，関数のインタフェースとして出力する．

「外部変数分類定義」の内容は外部変数をどのようにグループ化するかを示したものである．テキストファイルとして記述されており，その形式は，1 行が一つの外部変数グループに対応し，各行にはグループの名称と属する外部変数の名称がタブコードで区切られ列挙されているというものである．支援ツールは，「外部変数分類定義」に基づいて，外部変数の名称を所属するグループ名称に置き換え，外部変数を構造体として統合した場合のシミュレーション結果を出力する．

なお，下位の関数が参照した外部変数も引数の形で（+記号により）区別して表示できる．また，関数の呼出し関係を字下げにより示す．既に出力した関数以下は階層を展開して出力するか，あるいは [既] マークを付けて省略するかを指定できる．なお，再帰処理の場合は警告のために [再] マークを表示する（図 4）．本ツールは，組込みソフトウェアのプロジェクトの一環として開発したので，高い信頼性，短期間，低コストの制約を満たすように設計した．

- 不具合の生じやすい構文解析に TAG 生成ツールを流用した．
- 短期間で開発するために，スクリプト言語を用いた．
- 上記のツールのコストを削減するためにオープ

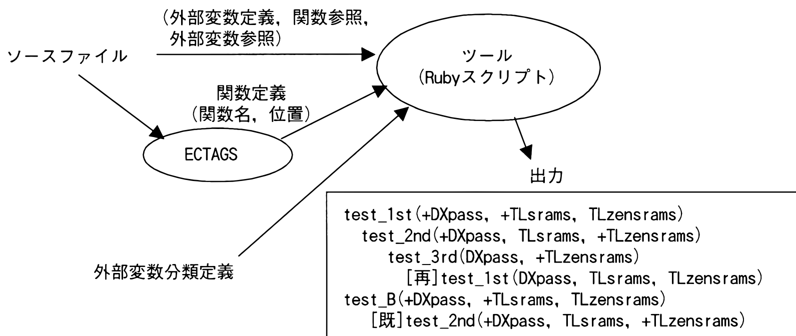

図 4 ツールのデータフロー
Fig. 4 Data flow of the tool.

ンソースのツールである ECTAGS[8]，Ruby[12] をそれぞれ用いた．

本ツールの機能を実現するには，関数の定義，外部変数の定義，関数の参照，外部変数の参照の情報をソースファイルから得る必要があるほか，外部変数をどのように統合するかを定義した外部変数の分類の定義が必要である．図 4 に示すように関数の名前と位置の定義を ECTAGS から，外部変数の分類の定義を外部変数分類定義ファイルからそれぞれ入力とし，ソースプログラムを部分的に構文解析して外部変数の参照と関数の参照を検索する，というスクリプトを Ruby で記述した．C 言語の構文のうち関数名とそのファイル上の位置がわかれば，

- 関数定義の抽出に必要な C 言語の多くの構文の解析が不要になる．
- 個々の関数のコード上で関数名を検索すれば関数の参照情報が得られる．

となり，ECTAGS の出力により，関数に関する構文解析の工数の多くを軽減することができた．なお，外部変数の定義は extern 宣言文を解析するだけで容易に抽出でき，参照情報は容易に抽出できた．ECTAGS の利用と Ruby により，開発規模は約 1000 ステップ程度になり，約 1 人月で作成できた．

4.2 ツールの導入

利用者の経験度を問わずソフトウェア改良法を短時間で習得できるドキュメントを整備した．改良法に関しては，設計の指針を示した設計ガイドを作成したほか，具体的なソースをもとに改良法の勉強会を主要なメンバーで実施した．設計ガイドには **3.2** で述べた内容のほか，**2.1** の記述も行い，本ツールの使用方法を中心になるべく一般的な内容で作成した．また，改良後の品質を維持できるように，設計とコーディングのチェックリストを，文献に基づいてそれぞれ作成した．既存の文献のチェック項目は，文献ごとに注目する内容が異なるため，チェック項目の統合は難しく，設計で 82 項目，コーディングで 98 項目になった．これらをレガシーソフトウェアに特有の内容，一般的な内容，基本的な内容の 3 種類に分類し，開発者の経験に合わせて，重要な部分だけを短時間で読めるようにした．

5. 結果と考察

5.1 試　　行

改良支援ツールを本格的に導入する前に試行を行い，ツールの効果を確認した．試行では，既にコードレビューの行われているプログラムをツールで評価（改良すべき点を指摘）し，コードレビューでの評価と比較した．

対象：　10 年以上改造が繰り返されてきた組込みソフトウェアで規模は約 5500 ステップ，試行前にプログラム構造を熟知したエンジニアが 8 時間かけてコードレビューした．

試行：　プログラム構造の知識のないエンジニアがツールの出力をもとに約 30 分間かけて評価した．

結果：　ツールの出力に基づき，図 3 に示した二つの「保守性の悪い状態」を次のとおり指摘することができた．ツールによる評価は，コードレビューに取って代わるものではないが，レビューでの指摘漏れを防止する効果は高いと考えられる．

(a)　「外部変数が多い」

モジュール構造図で最下位に位置する関数（単機能な汎用関数）を除いたほぼすべての関数で，参照される外部変数の数が多いことがわかった．この点は，コードレビューにおいても指摘されていた．なお，該当箇所を改良するためには多くの工数が必要となるため，必要性は認められるが実際には改良は行わない，とコードレビューでは判断されていた．

次に，ソースファイル上でのデータ種別の分類をもとに作成した「外部変数分類定義」を用いてシミュレーションを行った．その結果，外部変数の数が多い関数のうち三つの関数については，外部変数が多数のグループに分類される（構造体として統合しても，数多くの構造体を用意しなければならない）ため，「共通処理の抽出」が必要であると判断された．これら三つの関数は，コードレビューにおいてもその問題点が指摘されており，うち二つの関数は，処理が手順的であり改良（共通処理の抽出）が必要であると判断されていた．残る一つの関数は，プログラムの実装を考慮すると改良する必要はないと判定されていた．

(b)　「モジュール構造が複雑」

プログラム全体を鳥瞰した結果，再帰処理マーク（[再]）がツールにより付加されており，かつ，自己再帰でなく複数の関数で構成される部分をプログラム中に 1 箇所発見することで，改良に役立てることができた．単純な機能でない上位関数が共通ルーチンとして使われており，本来は，共通処理を抽出して共通関数を作成すべき部分である．しかし，コードレビューでは，改良すべき点としての指摘はなされていなかった．

5.2 導入結果

試行後に実際の開発作業に導入し，開発が最終テスト段階になった時点で，アンケートをとった．このツールは，レガシーソフトウェアを改造して新しいシステムを作るプロジェクトに導入された．導入にあたり，設計ガイドとチェックリストをもとに，より具体的な改良法の勉強会を中心メンバ 3〜4 人で 2 時間ずつ，5 回実施した．最終テスト段階のアンケートで，以下の意見が得られた．

- 改造元ソースの悪い構造が見えたので，もともと開発されたときと同じ構造にならないように参考にできた．
- 設計時に「保守性の高いソースを作る」という意識で開発・レビューできた．
- もとのプログラムに比べ，データを隠ぺいしたり，シンプルな構造にすることができた．その効果はその後の保守で実感した．
- 勉強会の実施により，知識が向上した．
- キャラクタインタフェースなので，あまり使わなかった．
- プログラム構造を大幅に変更することがこわかったので，直接コードを見てレビューをした．

5.3 オープンソースを用いた内製の効果

今回，オープンソースである Ruby と ECTAGS を用いたツールを内製することで，市販のツールにはない，以下の効果があった．

- 利用者が多いツールであり，信頼性が高かったので，ツールの欠陥によるトラブルがなかった．将来，欠陥が生じた場合も，対応策をとれる可能性がある．
- スクリプト言語を用いて内製したので，必要な情報を必要なフォーマットで得られ，仕様の追加などの保守も容易であった．
- 社内に展開する際もコストがかからない．
- Windows と UNIX の二つの環境で実現でき，クロス環境でも用いることができた．

5.4 考察

アンケートの結果を見ると，支援ツール導入の効果はあったが，担当する作業により評価は分かれていた．高い評価をしていたのはリーダであった．改良法がわかること，改良後の構造を確認できることから評価が高かった．従来は評価の方法論が確立していなかったので，レビューを行っても保守性を高められなかった．本研究では外部変数によるデータの受け渡しを含めたプログラム構造の評価法を提案し，新たなツールを開発し，ツールで支援することで，大きな負担をかけずにレビューができるようになった．一方，評価が低かったのはメンバであった．メンバにとって，今回のツールは悪いところを見つける手助けはするが，改良作業そのものを支援しないことから評価が低かった．開発されたツールは，ユーザインタフェースを改良することで，利用率が向上すると考えられるが，大幅な変更に対する不安を取り除くためには，実績を積むほか，XP のリファクタリングの技術を導入するなど改良作業そのものの支援の検討が必要と考えられる．

オープンソースのツールを用いて，支援ツールを内製することは信頼性，納期，コストの面で効果があった．オープンソースは無料であるというだけでなく，信頼性が高いことから用いられることが多く [21]，本研究で使用したツールにおいても信頼性は高かったといえる．また，変更が容易で，クロス環境が実現できたなど，自由度が高かった．ユーザインタフェースが使いにくいなど，必ずしもすべての要求を満たすものではなかったが，最低限必要とされる機能を，少ないコストで，柔軟な仕様で実現することができた．既存のツールでは対応していないようなツールが必要な場合，不具合の生じやすい部分になるべくオープンソースのツールを用いて，内製することが効果的であると考えられる．

6. むすび

レガシーソフトウェアを調査し，外部変数によるデータの受け渡しを含めたプログラム構造を評価する方法を提案した．また，オープンソースのツールを用いて支援ツールを作成した．このツールは，提案した評価法に基づき，外部変数によるデータの受け渡しを引数によるインタフェースとして表示するとともに，モジュール構造を表示するものである．従来レビューが困難であった，外部変数によるデータの受け渡しを含めたプログラム構造を示すことで，改良すべき点の検討や改良後のソフトウェアの確認に有効であった．また，オープンソースのツールを用いて内製することで，少ない工数で利用者の意見に合わせたツールを作ることができた．しかし，リーダだけでなくメンバが進んで使うようにするには，他の方法論との併用やユーザインタフェースの改良が必要である．現在，UNIX 及びウィンドウズで動作するフリーウェアである Tcl/Tk を用いて，図 5 に示すグラフィカルユーザインタフェースを開発する等の改良を進めている．

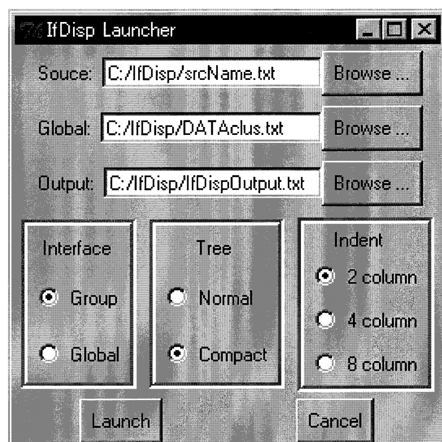

図 5 ツールのユーザインタフェース（GUI 版）
Fig. 5 The user interface of the tool (GUI version).

本研究の特徴は，組込みソフトウェアの信頼性，納期，コストの制約の中で，より保守性の高いソフトウェアを得る方法を示していることである．オブジェクト指向においても何らかの制約によりオブジェクト指向言語を用いることができない場合がある．このような場合は，論理設計をオブジェクト指向で行い，それを開発言語で写像する方法が用いられる [17]．このような方法は消極的ではあるが，方法論の適用できる範囲を広げることが可能である．本研究も組込みソフトウェアの制約の中で，既存の資産を最大限に生かしながら実施することができる．これまで，多くの制約の中で大規模な改良をあきらめざるを得なかったソフトウェアであっても，保守性を高めることができる可能性がある．

また，組込みソフトウェアの特徴は最近のアプリケーションの特徴でもあり，今回の成果が生かせる可能性がある．外部変数は構造化設計やオブジェクト指向設計の普及により，少なくなってきている．しかし，最近のアプリケーションにおいても，終了時の情報を不揮発メモリであるレジストリに保存しておき，再び実行された際に，前回の続きから実行できるように作られる場合が多い．このようなアプリケーションでは，システム関数を介しているが，実質的にレジストリのデータを直接操作している場合が多い．この場合のレジストリは，組込みソフトウェアにおける外部変数と同じ使い方であると考えられる．レジストリが複雑になり，プログラムの保守性が悪くなったプログラムに，今回の方法論とツールを適用できる可能性がある．

文　献

[1] 青木　淳，オブジェクト指向システム分析設計入門，p.16, ソフト・リサーチ・センター，1993.

[2] J.D. Ahrens and N.S. Prywes, “Transition to a legacy- and reuse-based software life cycle,” IEEE Computer, vol.28, no.10, pp.27–36, 1995.

[3] K. Beck, Extreme Programming Explained: Embrace Change, pp.21–25, pp.107–108, Addison-Wesley, 1999.

[4] G. Booch, Object-oriented analysis and design with applications, Rational, 1994, 山城明宏，井上勝博，田中博明，入江　豊，清水洋子，小尾俊之訳，Booch 法：オブジェクト指向分析と設計 第 2 版，pp.57–63, アジソン・ウェスレイ・パブリッシャーズ・ジャパン，1979.

[5] G. Canfora, A. Cimitile, and U. de Carlini, “An extensible system for source code analysis,” IEEE Trans. Software Eng., vol.24, no.9, pp.721–740, 1998.

[6] L. Cleveland, “A program understanding support environment,” IBM Syst. J., vol.28, no.2, pp.324–344, 1989.

[7] 長谷川邦夫，事例：日本ユニシスにおける保守管理の実際，ソフトウェアプロジェクト管理 上，菅野文友監修，pp.525–549, 1990.

[8] D. Hiebert, EXUBERANT CTAGS, http://home.hiwaay.net/~darren/ctags/, 1999.

[9] 平野一路，“遺産的 (Legacy) システムを対象にした進化的保守開発，” 第 16 回ソフトウェア信頼性シンポジウム論文集，pp.16–19, 1996.

[10] G. Lenford and J. Myers, Composite / structured design, Litton Educational Publishing, 1978, 国友義久，伊藤武夫訳，ソフトウェアの複合/構造化設計，pp.61–67, 近代科学社，1979.

[11] W.C. Lim, “Effects of reuse on quality, productivity, and economics,” IEEE Software, vol.11, no.5, pp.23–30, 1994.

[12] まつもとゆきひろ，オブジェクト指向スクリプト言語 Ruby，アスキー，1999.

[13] 宮本　勲，ソフトウェア保守の管理，p.15, TBS 出版会，1984.

[14] 内藤裕幹，飯田　元，松本健一，鳥居宏次，“役割別工数投入計画のための見積もりモデルの提案，” 情処学ソフトウェア工学研報，vol.96, no.6, pp.1–8, 情報処理学会，1996.

[15] 中本幸一，高田広章，田丸喜一郎，“組込みシステム技術の現状と動向，” 情報処理，vol.38, no.10, pp.871–878, 情報処理学会，1997.

[16] M. Page-Jones, Practical guide to structured system design second edition, Prentice-Hall, 1988, 久保未沙，新谷勝利訳，構造化システム設計への実践的ガイド，pp.57–142, 近代科学社，1991.

[17] J. Rumbaugh, Object-oriented modeling and design, Prentice-Hall, 1992, 羽生田栄一訳，オブジェクト指向方法論 OMT，pp.310–311, 371–397, トッパン，1992.

[18] 阪井　誠，久保田益史，沖田昌也，松本健一，鳥居宏次，“レガシーな組込みソフトウェア改良のための支援ツール，”

ソフトウェア・シンポジウム'99 論文集，pp.59–66, 1999.
[19] H.M. Sneed, "Economics of software re-engineering," J. Software Maintenance: Research and Practice, vol.3, no.3, pp.163–182, 1991.
[20] 高橋良英，中村行宏，"流用率と流用部・改造部間のインタフェースの複雑さによるソフトウェアの保守品質評価方法，" 信学論（D-I），vol.J80-D-I, no.5, pp.441–449, May 1997.
[21] 山中　勝，"オープンソースとしてのリアルタイムシステム，" ソフトウェアシンポジウム 2000 論文集，pp.55–59, 2000.

（平成 12 年 8 月 28 日受付，12 月 18 日再受付）

阪井　　誠　（正員）

昭 59 大阪電通大・工・電子機械卒．同年 SRA に入社．以来，ソフトウェア開発・研究開発に従事．平 13 奈良先端科学技術大学院大学博士後期課程了．同年 SRA 先端技術研究所シニア研究員．博士（工学）．情報処理学会，IEEE，ソフトウェア技術者協会各会員．

久保田益史

昭 59 阪府大・工・経営卒．昭 61 同大大学院工学部修士課程了．同年オムロンに入社．以来，ソフトソフトウェア開発・研究開発やソフトウェア開発のプロジェクトマネジメントに従事．現在オムロンソーシャル事業グループ開発・生産センタソフトウェア開発部主査．

沖田　昌也

平 2 東京農工大・工・数理情報卒．同年オムロンに入社．主にソフトウェア開発に従事．平 13 UC Berkeley Extension 了．現在オムロンソーシャルシステムズビジネスカンパニー EFTS 統轄事業部開発製造センター主事．

松本　健一　（正員）

昭 60 阪大・基礎工学・情報卒．平元同大大学院博士課程中退．同年同大基礎工学部情報工学科助手．平 5 奈良先端科学技術大学院大学助教授．平 13 同大教授．工博．Computer-Aided Empirical Software Engineering (CAESE) 環境，ソフトウェアメトリクス，ソフトウェアプロセス，視線インタフェースに関する研究に従事．IEEE，ACM 各会員．

鳥居　宏次　（正員）

昭 37 阪大・工・通信卒．昭 42 同大大学院博士課程了．同年電気試験所（現電子技術総合研究所）入所．昭 59 大阪大学基礎工学部情報工学科教授．平 4 奈良先端科学技術大学院大学教授．平 11 同大副学長．平 13 同大学長．工博．専門はソフトウェア工学．IEEE, ACM 及び情報処理学会の各フェロー．